Solving Sparse Triangular Linear Systems Using FORTRAN with Parallel Extensions on the NYU Ultracomputer Prototype

by

Anne Greenbaum

Ultracomputer Note #99
April, 1986

ABSTRACT

A parallel method is presented for solving sparse triangular linear systems. Several implementations of this algorithm using FORTRAN with parallel extensions on the NYU Ultracomputer Prototype are described. Timing results are given, along with analysis of expected speedup. Possible techniques for improving performance are also discussed.

1. Introduction

In the numerical solution of partial differential equations, a frequently encountered problem is the solution of large sparse triangular systems of linear equations. For example, when solving elliptic and parabolic partial differential equations, one frequently uses an iterative method with a preconditioning matrix. The preconditioning matrix may have the same nonzero structure as the lower triangle of the matrix being solved (Gauss-Seidel, SOR methods), or it may be given by the product of a lower and an upper triangular matrix, each of which has approximately the same nonzero structure as the lower or upper triangle of the matrix being solved (symmetric Gauss-Seidel, SSOR, incomplete Cholesky, and incomplete LU methods). At each iteration, then, solutions to one or more triangular linear systems must be computed. Techniques for solving hyperbolic partial differential equations often involve marching through a grid in the direction of characteristics, computing the solution at a given node once the solution is known at the other nodes of that zone. This can also be thought of as solving a triangular system of linear equations, with couplings between each node and the earlier numbered nodes of the same zone.

Solution of such triangular systems is often a bottleneck on vector machines, because the process can at most be partially vectorized. If the grid structure is regular, then solution components that can be computed simultaneously are normally stored with a constant vector stride. In this case, the solution can be performed in several sequential stages, each of which is vectorized, with the vector length varying as the number of components that can be computed changes. If the grid structure is irregular, however, it may be difficult or impossible to store the matrix and solution vector in such a way that the computation can be vectorized directly. It may be necessary to use GATHER/SCATTER operations before invoking vectorization. If the number of points to which different nodes are coupled varies, then different operations will be necessary to compute different solution components. In this case, vectorization becomes almost impossible. In all of these cases, however, parallel computation of certain solution components is straightforward.

A program has been written to search through the couplings of a sparse triangular matrix and determine which solution components can be computed simultaneously. A parallel backsolving routine then performs this computation. This program has been implemented on the NYU Ultracomputer Prototype, using FORTRAN with extensions for parallelism. Several different styles of parallel programming have been tried. Details of the parallel implementations, along with timing results, are presented in this paper. Other implementation strategies are being planned and will be described in a later report.

2. The Problem.

Consider solving a triangular linear system $Tx=b$, where the triangular matrix T has the following form:

```
row
1       x
2       x  x
3          x  x
4          x     x
5             x  x  x
6       x              x
7       x  x              x
8          x  x              x
 col    1  2  3  4  5  6  7  8
```

Here the x's represent nonzero elements of T and the other elements are assumed to be zero. The solution can be accomplished in four stages:

First use equation 1 to solve for the first solution component.

Given this first component, solve either equation 2 for the second component or equation 6 for the sixth component, since these couple only to component 1. Equations 2 and 6 can be solved in parallel.

Having computed components 1, 2, and 6, then, equations 3, 4, and 7 can be solved simultaneously to give components 3, 4, and 7.

Finally, with these components known, equations 5 and 8 can be solved in parallel for components 5 and 8.

A program has been written to search through the couplings of a sparse triangular matrix, as above, and determine which solution components can be computed in parallel. Details of this code will not be given here, as more work needs to be done to make it more efficient. Given the list of components that can be computed at each stage, a parallel backsolving routine carries out this computation. It is this backsolving routine and its implementation on the Ultracomputer Prototype that is described in the following sections.

3. Implementations of Parallel Backsolving.

The FORTRAN code for backsolving of a unit lower triangular linear system Tx=b, in an order amenable to parallelization, can be written as follows:

```
C  First store right hand side b in solution vector x.
      DO 10 I=1,N
10      X(I) = B(I)

C  Now loop over sequential stages.
C  Array LIST contains a list of all equations, in the order
C    in which they can be solved;  e.g., 1, 2,6, 3,4,7, 5,8
C    in the previous example.
C  Array ILIST points to the first equation of each
C    stage in LIST and to the next position after the last
C    stage; e.g., 1, 2, 4, 7, 9  in the previous example.
       DO 40 KSTAGE=1,NSTAGE
         M1 = ILIST(KSTAGE)
```

```
          M2 = ILIST(KSTAGE+1) - 1

C     Parallel section.  Loop through all equations that
C     can be solved at this stage.

C     The nonzeros of the triangular matrix T are stored
C       by rows, with the first element of each row being
C       the unit diagonal element.
C     Array IT points to the beginning of each row in T
C       and to the next position after the last element
C       of the last row.
C     Array JT contains the column number of each element of T.

          DO 30 M=M1,M2
            I = LIST(M)
            K1 = IT(I) + 1
            K2 = IT(I+1) - 1
            IF(K1 .GT. K2) GO TO 30
            DO 20 K=K1,K2
20            X(I) = X(I) - T(K) * X(JT(K))
30          CONTINUE

C     End of parallel section.


40        CONTINUE
```

3.1. An Easy Way to Parallelize.

Probably the easiest way to parallelize loop 30 on the Ultracomputer is to use only the two primitives SPAWN and MWAIT. SPAWN creates multiple processes, and MWAIT waits for them to complete. For details on the use of these and other multitasking primitives on the Ultracomputer Prototype, see [1] and [2]. Each time through loop 40, we can compute the number of equations that can be solved at this stage, M2-M1+1, and spawn M2-M1 children which, along with the parent, can perform the work in loop 30. Each child process stops when its work is complete, while the parent calls MWAIT to wait until all children have terminated. This process is then repeated for each pass through loop 40. The coding for this is as follows:

```
         DO 40 KSTAGE=1,NSTAGE
           M1 = ILIST(KSTAGE)
           M2 = ILIST(KSTAGE+1) - 1

C  Compute number of children NCHLD equal to number of
C  equations minus one.  Each child will solve one equation,
C  as will the parent.  The equation to be solved is determined
C  by the process id ICHLD, returned by SPAWN.
           NCHLD = M2 - M1
```

```
      ICHLD = 0
      IF(NCHLD .NE. 0) ICHLD = SPAWN(NCHLD, 0, STATUS, NCHLD+1)
      M = M1 + ICHLD
C     Each process does its piece of the work.
        I = LIST(M)
        K1 = IT(I) + 1
        K2 = IT(I+1) - 1
        IF(K1 .GT. K2) GO TO 35
        DO 20 K=K1,K2
20        X(I) = X(I) - T(K) * X(JT(K))
C     Children now terminate, parent waits for all children to
C     terminate.
35      IF(ICHLD .NE. 0) STOP
        IF(ICHLD .EQ. 0 .AND. NCHLD .NE. 0) IWAIT = MWAIT(0)

40    CONTIUNE
```

In the above code, arrays X, IT, JT, T, and LIST must reside in shared memory, so that all of the processors have access to them. For details on how to do this, see [1]. Variables M, I, K1, K2, and K are local variables, with each process having its own copy of them.

While the above implementation requires little modification to the original FORTRAN code, it does have some drawbacks. Spawning tasks is a time-consuming process, requiring interaction with the operating system. Unless each task has a large amount of work to do, one may spend more time spawning the tasks than actually computing!

This code was tried on a small test problem. The triangular matrix was taken to be the lower triangle of a five-point approximation to the Laplacian on a 10 by 10 rectangular grid. The order in which solution components can be computed is illustrated in Fig. 1. The first time through loop 40, there was only one equation to solve. The next time through, there were two. The next time, three, etc. Up to a maximum of ten. The number of equations then decreased again: nine, eight, ... down to one. The solution of each equation required two multiplications and additions.

Because of the small granularity of the problem -- each process had only two additions and multiplications to perform -- the parallel code was much slower than the serial version. The time for spawning a task is considerably greater than the time for performing two additions and multiplications!

To increase the granularity, a parameter ICHUNK was added to specify how many equations each process should solve. That is, instead of having each process solve only one equation, there will be fewer processes, each solving ICHUNK equations. A reasonable value for ICHUNK is one for which the time to spawn a task is considerably less than the time for a single processor to solve ICHUNK equations. The parallel code is then modified as follows: NCHLD is set to (M2-M1)/ICHUNK; instead of having each process work with a single value of M, each process computes M1P=M1+ICHLD*ICHUNK, M2P=MIN0(M1P+ICHUNK-1,M2), and loops over the work section with M going from M1P to M2P.

For the small test problem mentioned above, it was found that the optimal value for ICHUNK was ICHUNK = 10; i.e., no tasks were spawned, the code was executed

serially! This indicates that the time for spawning a task on the Ultracomputer prototype is large compared to the time for a single processor to solve ten equations, or, to perform 20 additions and multiplications. This is not surprising. By using larger test problems and thereby allowing larger granularity, we can eventually find the size problem for which the above code will execute efficiently; i.e., will provide almost the maximum possible speedup.

3.2. A More Efficient Way to Parallelize.

Rather than creating large granularity through large problems, however, one might try to improve the style of parallel programming to make the code work efficiently on smaller problems. To this end, we tried a different implementation of the parallel backsolving routine. The coding becomes somewhat more complicated now. In addition to the routines SPAWN and MWAIT, the barrier routines BWBARI and BWBARR will also be used. These are also described in [2]. Rather than spawning children as they are needed, since this takes so long, we will spawn the maximum number of children at the start and have each of them go through loop 40 computing NCHLD = (M2-M1)/ICHUNK at each iteration. If the id number of the child is greater than NCHLD, then it jumps over the work section and calls BWBARR to signal that it has completed its work (since it had none) and to wait for the other children and parent to complete theirs. If the id number of the process is less than or equal to NCHLD, then this process performs its piece of the work as before, and then calls BWBARR. The coding for this version of the program is as follows:

```
C   Compute maximum number of children.
      NCMAX = 0
      DO 4 KSTAGE=1,NSTAGE
        M1 = ILIST(KSTAGE)
        M2 = ILIST(KSTAGE+1) - 1
        NCHLD = (M2-M1)/ICHUNK
        IF(NCHLD .GT. NCMAX) NCMAX = NCHLD
4       CONTINUE
C   Initialize barrier and spawn maximum number of children.
      NPROC = NCMAX + 1
      IDBARR = BWBARI(NPROC)
      ICHLD = 0
      IF(NCMAX .NE. 0) ICHLD = SPAWN(NCMAX, 0, STATUS, NCMAX+1)
C   Send parent and each child through loop 40.  If process has no
C   work to do at this stage, let it skip work section and jump
C   to barrier.
      DO 40 KSTAGE=1,NSTAGE
        M1 = ILIST(KSTAGE)
        M2 = ILIST(KSTAGE+1) - 1
        NCHLD = (M2-M1)/ICHUNK
        IF(ICHLD .GT. NCHLD) GO TO 35
          M1P = M1 + ICHLD*ICHUNK
          M2P = M1P + ICHUNK - 1
          IF(M2P .GT. M2) M2P = M2
```

```
            DO 30 M=M1P,M2P
              work section

30            CONTINUE
35          CALL BWBARR(IDBARR)
40          CONTINUE

C   Terminate children.
        IF(ICHLD .NE. 0) STOP
```

4. Timing Tests.

This code was also tried on the test problem mentioned previously, as well as on similar problems arising on a 20 by 20 grid and on a 30 by 30 grid. Timing results for the serial code, the parallel version using only SPAWN and MWAIT (par1), and the parallel version using BWBARR (par2) are shown in Table 1.

The timers for the serial code and the first parallel version were put around loop 40. Thus, the timings for par1 include the overhead for calling SPAWN and MWAIT many times. The timers for the second parallel version were invoked AFTER the initial spawning of the maximum number of children and were terminated when loop 40 completed. This is reasonable if we assume that children already would have been spawned, for other purposes, and thus would be available for the parallel backsolving.

Tests were run in standalone mode, and the backsolving routine was called approximately twenty times during each run. There was some variation in the timings obtained on different calls, and these are reflected in Table 1, which gives the ranges of times obtained. Because the clocks used are accurate only to within 1/60 sec. for the user + system time and to within 1 sec. for the wall clock time, a variation of this size is meaningless. The column labelled "user + system time" represents only the PARENT's time. It is the sum of the first two numbers returned by routine "times" [2]. The parent's work time should be greater than or equal to the work time of any child process, since the parent has the most computations to perform and must wait for all of the children to complete, anyway. The time for spawning tasks, however, may appear only partially in the parent's time. This is indicated by the large difference between the "user + system time" returned for par1 and the actual "wall clock time". It is the *wall clock time* that is important in this case. For the serial code and for par2, the user + system time and the wall clock time are equal, to within the accuracy of the timers. The column labelled "wall clock time" is the number returned by routine "time" [2].

On all problems tried and for all values of ICHUNK that required spawning of children, the first parallel version of the code was slower than the serial version. (Not all cases were even run with this version of the code, since it was clear that it would not provide speedup over the serial version.) Of course, for ICHUNK large enough, no children are spawned, and then all three codes are equivalent. The second parallel version, however, obtained significant speedup over the serial version. The best speedup was about a factor of 3. The program was run using both the error-checking version of BWBARR in -lP77 and the faster version in -lP77_f [2]. No significant difference in total time was observed. Since there were 6 processors available, perfect parallelism should result in a speedup of about a factor of 6. Note, however, that for the type of problem being solved, we do not have perfect load balancing and cannot use all of the processors all of the time.

In the following section, we show that a factor of 3 speedup using 6 processors is about the best we can expect on a problem of this sort, using static parallelism as described.

grid size	ICHUNK	#processes	user + system time (1/60 seconds)			wall clock time (seconds)		
			serial	par1	par2	serial	par1	par2
10 by 10	2	5		8-27	3-4		2-3	0-1
	3	4		3-12	3-4		2-3	0-1
	4	3		11-22	7-8		1-2	0-1
	5	2		11-23	7-8		1-2	0-1
	10	1	11	7-12	10-16	0-1	0-1	0-1
20 by 20	4	5			15			0-1
	5	4			18-19			0-1
	10	2		42-56	33-34		3	0-1
	20	1	41-45		45	0-1		0-1
30 by 30	5	6			33-34			0-1
	30	1	97-102		97-101	1-2		1-2

Table 1. Timing Results for Parallel Backsolving.

Before analyzing the potential speedup for this problem, we should point out that one other version of the parallel code was implemented and timed. This version spawned the maximum number of children needed and simply "put them to sleep" until they were needed. That is, only the parent task went through loop 40 and computed the number of children NCHLD needed at each stage. The children spun in a loop, testing to see if the value of NCHLD was greater than or equal to their task id. If it was, the child did its piece of the work and then called BWBARR, but the parent had called BWBARN so that only NCHLD+1 processes synchronized at the barrier, instead of all NCMAX+1 of them. Because the Ultracomputer prototype has processors connected to shared memory via a bus, this version should be faster than par2, if there are many processors. With the six processor prototype, however, no difference in timing was observed between the two codes. When the bus is replaced by an Omega network, there should be little difference in the two codes, even if large numbers of processors are being used.

5. Analysis and Future Directions.

Suppose we have a problem, like the test problems of the previous section, for which we can perform one unit of work at the first stage, two at the second, three at the third, and so on, up to N. We then perform N-1 units of work at the next stage, N-2 at the next, and so on, back down to one again. Let P be the number of processors to be used and assume, for convenience, that P divides N. As in the test problems, a new process will be assigned work only when the amount of work at a particular stage exceeds $\frac{N}{P}$ times the number of processes currently being used.

The time to complete the first $\frac{N}{P}$ stages is

$$1 + 2 + \cdots + \frac{N}{P} = \left(\frac{N}{P}\right) * \frac{\left(\frac{N}{P}+1\right)}{2}.$$

Each successive stage, up to stage N, requires time $\frac{N}{P}$, as do the next $(N-1)-\frac{N}{P}$ stages. The final $\frac{N}{P}$ stages require the same amount of time as the first. The total time, then, is equal to

$$\left(\frac{N}{P}\right) * \left[\left(\frac{N}{P}+1\right) + 2 * \left(N-\frac{N}{P}\right)-1\right].$$

When P is 1, so the code is executed sequentially, this expression is simply N * N.
Hence, the expected speedup, using P processors instead of 1, and ignoring any overhead costs for synchronizing processors, etc., is given by

$$P * \frac{1}{2-\frac{1}{P}}. \qquad (1)$$

If P is large, then the speedup predicted by (1) is approximately $P * \left(\frac{1}{2}\right)$.

Using this formula, and a slightly modified version when P does not divide N, we can estimate the maximum possible speedup for the test problems. These numbers are shown in Table 2, along with the actual speedup, derived from the user+system time clock, for par2. Because there was some variation in the clock readings on different runs, the speedup is taken to be the ratio of the shortest measured time for the serial code to the shortest measured time for the parallel code.

grid size	ICHUNK	#processes	maximum speedup	measured speedup for par2
10 by 10	2	5	2.8	3.7
	3	4	2.0	3.7
	4	3	1.6	1.6
	5	2	1.3	1.6
20 by 20	4	5	2.8	2.7
	5	4	2.3	2.3
	10	2	1.3	1.2
30 by 30	5	6	3.3	2.9

Table 2. Maximum Speedup and Measured Speedup for Par2.

While the predicted and measured speedups agree very well on many of the problems, there is some difference on the 10 by 10 case. It appears that the parallel code is getting better than the maximum possible speedup. This could be caused by several factors. One

is the inaccuracy in the clock. For such short time intervals, a difference of even one 1/60-second may cause a significant change in the computed speedup. On the 10 by 10 problem with ICHUNK equal to 2 or 3, for example, if we use the longest measured time (4/60 secs) instead of the shortest measured time (3/60 secs) for the parallel code, then the computed speedup changes from 3.7 to 2.75. This is still somewhat higher than predicted, however, at least for the ICHUNK=3 case, and it is expected that there is something more going on. Another possible explanation is operating system interference. If even the shortest measured time for the serial code includes some operating system interference, and if the time for the parallel code does not, this could account for the extra speedup. A third possibility is caching. Certain quantities in the parallel code may be stored in cache, while the corresponding quantities in the serial code are not. These possibilities are currently being investigated and will be discussed further in a later report.

Recall that the parameter ICHUNK used in the test problems was introduced in the first version of the parallel code, par1, because the time for spawning a task was large compared to the time for solving a single equation. This parameter allowed more equations to be solved between calls to routine SPAWN. In the second parallel version, par2, however, the overhead for putting an already spawned process to work is practically nil. If the process is not assigned any work, it jumps over the work section and calls BWBARR, where it has to wait for those processes that are doing work, anyway. It could just as easily do a piece of the work itself. Thus, for this version, the optimal value for ICHUNK should be ICHUNK=1.

There is one problem with this, and that is that we will be spawning more processes than there are processors available. Since all processes will be busy-waiting on the others, one might suspect the possibility of deadlock. The scheduler on the Ultracomputer Prototype protects against this by occasionally swapping jobs out of the CPU's so that others can get in. While the program will not deadlock, it may be very slow to execute, depending on the frequency at which jobs are swapped in and out. A non-busy-waiting barrier (one that automatically gives up the CPU while waiting) would be more appropriate in this case, but this would again entail significant overhead. This type of barrier routine is not currently available on the Ultracomputer prototype. Another way to carry out the process is to do one's own scheduling.

To this end, we modified the second parallel code, par2, as follows: Instead of spawning NPMAX children, if this number is greater than 5, we spawn only 5 children (so the total number of processes is no greater than 6, the number of processors). Just before statement 35, the call to BWBARR, a statement was added to check if NCHLD is greater than 5. If it is, it is decremented by 6, M1 is replaced by M1+6*ICHUNK (since the first 6*ICHUNK equations will be solved in the first pass through the loop), and the process jumps back to the statement IF(ICHLD .GT. NCHLD)... . If the process id, ICHLD, is still less than or equal to NCHLD, then the process does another piece of work, etc. Assuming all processors complete a fixed size piece of work at the same time, then, this makes maximal use of the processors in solving the equations at each stage.

The expected speedup, ignoring overhead costs, etc. for this type of parallelism can be estimated as follows. Again, assuming that there are P processors and a maximum of N units of work, it can be seen that the first P stages each require one unit of time, the next P stages each require 2 units of time, etc. The total time for the first N stages is given by

$$P * 1 + P * 2 + \cdots + P * \left(\frac{N}{P}\right) = N * \frac{\left(\frac{N}{P}+1\right)}{2}.$$

The time for the remaining N-1 stages is

$$(P-1) * \left(\frac{N}{P}\right) + P * \left(\frac{N}{P}-1\right) + \cdots + P * 1 = N * \left[\frac{\left(\frac{N}{P}+1\right)}{2} - \frac{1}{P}\right]$$

giving a total time of

$$N * \left(\frac{N}{P} + 1 - \frac{1}{P}\right)$$

for the entire computation. When P is 1, this again is just N * N, and taking the ratio of this to the expression for general P gives a predicted speedup of

$$P * \frac{1}{1+\frac{P-1}{N}}. \qquad (2)$$

If N is very large compared to the number of processors P, then the idle time of any processor will be much less than its work time, and almost a full factor of P speedup should be attainable. If N is just a small multiple of P, however, then the expected speedup approaches a lower bound of $P * \left(\frac{1}{2}\right)$, which is essentially attained if P equals N and N is large.

Table 3 shows timings for the modified version of par2 with ICHUNK equal to one, along with the measured speedup over the serial code (again taken to be the ratio of the shortest measured time for the serial code to the shortest measured time for the parallel code), and the maximum possible speedup, obtained from formula (2) or from a modified formula when P does not divide N.

grid size	N	P	user + system time (1/60 seconds)	measured speedup	maximun speedup
10 by 10	10	6	3-4	3.7	3.8
20 by 20	20	6	11-12	3.7	4.8
30 by 30	30	6	22-23	4.4	5.1
40 by 40	40	6	37-38 (180-184 serial)	4.9	5.3

Table 3. Measured and Maximum Speedup for Modified Par2 with ICHUNK=1.

The measured speedups in Table 3 are not quite as good as the maximum speedups calculated analytically, though for larger problems (with larger granularity), they seem to be approaching this maximum. This may be due to the static nature of the scheduling of processes, causing unnecessary idle time. At each stage, each processor solves a predetermined set of equations based on its process id number. While work is divided as evenly as possible among the processors, some may finish sooner than others. If there are, say, seven equations to solve at some stage, it would seem more reasonable to have each of the six processors solve one equation and then assign the seventh equation to whichever processor finishes first. Similarly, with larger numbers of equations, work would be assigned to the processors dynamically, as they become available. This dynamic assignment of resources should reduce idle time but at the cost of requiring some extra overhead for keeping track of which pieces of work remain to be assigned. This is planned as the next experiment.

A greater improvement along these lines may come from not dividing the work into strict stages with no overlap. Some of the equations at a given stage may not require all of the solutions from a previous stage. In the example problem described at the beginning, for example, not all equations to be solved at stage 3 require solutions from all equations at stage 2. (In fact, none of them do.) Components 3 and 4 can be computed as soon as component 2 is known, and component 7 can be computed as soon as components 1 and 2 are known (which is equivalent to saying as soon as component 2 is known). If equation 2 is solved sooner than equation 6 in the second stage, there is no reason why available processor(s) could not begin work on the third stage, and similarly, as soon as equation 3 of the third stage is solved, equation 8 of the fourth stage could be solved, etc. This amounts to a dataflow type of parallelism, in which each component is computed as soon as the needed inputs are available. While decreasing idle time of the processors, there may be significant overhead for keeping track of the equations that have and have not been assigned to processors. This is also planned as a future experiment.

References.

[1] Albert Cahana, Jan Edler, and Edith Schonberg. "How to Write Parallel Programs for the NYU Ultracomputer Prototype". Ultracomputer Documentation Note #2, Revision 1.11 (March 20, 1986).

[2] "NYU Ultracomputer UNIX Programmer's Manual". Documentation Note #1, (March, 1986).

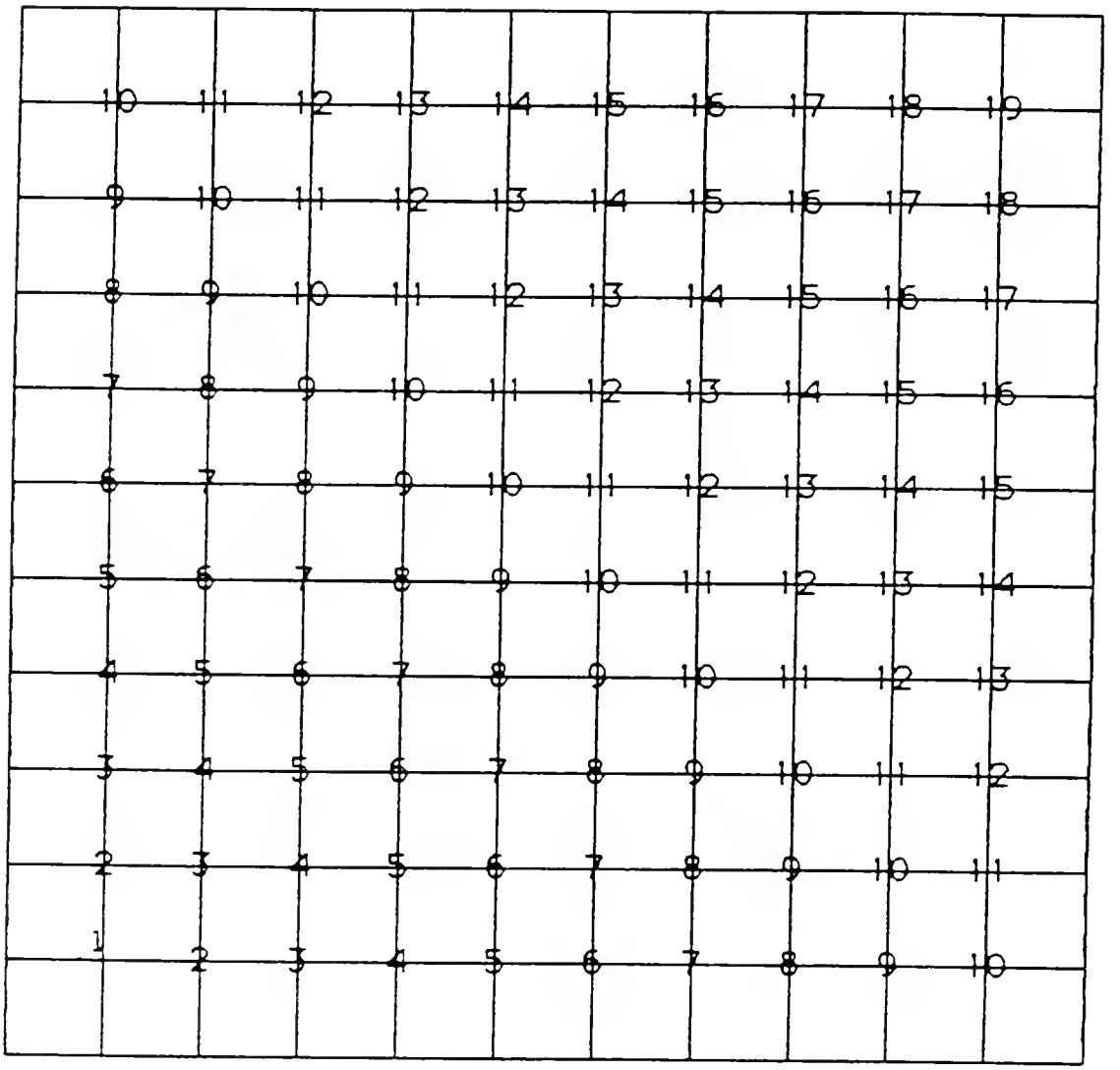

Fig. 1. Order of Solution of Points on a 10 by 10 Grid When Backsolving the Lower Triangle of a Five-Point Laplacian Operator.